OSU-RÉUNION | Observatoire des Sciences de l'Univers de La Réunion

Note technique

Guide pour le développement de logiciels

Référence : informatique/logiciel/nt_ums3365_guide-developpement-logiciels

Auteur : F. Gabarrot, R. Decoupes, G. Payen

Version du document : b.0

Première édition : 31/07/2014

Dernière révision : 28/02/2015

Nombre de pages : 22

Historique des évolutions	
31/07/2014	Création.
19/12/2014	Re-lecture et correction des fautes d'orthographe.
28/02/2015	Re-écriture du document et compléments.

Table des matières

1. DOCUMENTATION

1.1 Documents de référence

Code	Référence	Titre	Auteurs	Date
DR01	/gestion-information/devspot/nt_ums3365_memo-utilisation-git-gitlab_devspot_va0	Mémo d'utilisation de Git et GitLab sur devspot.	F. Gabarrot	2014/07/28

1.2 Sites internet

Code	Référence	Titre	Auteurs	Date
ST01	http://python.developpez.com/cours/TutoSwinnen/	Adaptation libre de "How to think like a computer scientist" de Allen B. Downey, Jeffrey Elkner et Chris Meyers	Gérard Swinnen	-
ST02	http://http://sametmax.com	Blog Sam&Max sur python entre autre	-	-
ST03	http://franckh.developpez.com/tutoriels/outils/doxygen/	DOxygen	-	-

2. INTRODUCTION

Ce document est un recensement de nos pratiques et de nos règles de fonctionnement pour le développement de logiciels.

Il a pour objectif de faciliter le travail en collaboration et de nous permettre d'échanger et de perfectionner ces pratiques et ces règles. N'hésitez pas à donner vos idées et à remplir les parties en attentes.

Pour commencer, voilà notre liste de clés à garder en tête pour le développement de logiciels :

- Prévilégier les langages C++ et Fortran pour les langages compilés et Python et Bash pour les langages interprétés. Mais sans se bloquer non plus !
- Prévilégier les outils libres et gratuits : gcc, gfortran, eclipse, geany, etc.
- Ne pas sous-estimer la phase de conception mais ne pas s'y noyer non plus : be Agile ! (http://fr.wikipedia.org/wiki/M%C3%A9thode_agile) … UML2, allons-y « ti lamb ti lamb » (Lucidchart pour les schémas c'est bien pratique aussi).
- S'assurer que les fonctionalités développées font le job et qu'elles sont les plus simples possibles à utiliser et à re-exploiter : KISS (http://fr.wikipedia.org/wiki/Principe_KISS).
- Assurer la gestion du code en configuration : Git et GitLab sur le devspot (voir DR01).
- Organiser les paquets logiciels de façon la plus standardisée possible pour que le code soit facilement reprenable (voir chapitres suivants : *rien n'est figé, chacun à le droit d'apporter des évolutions*).
- Documenter tous les codes afin qu'ils soient facilement utilisables et re-exploitables :
  - ➔ commentaires dans les fichiers sources ;
  - ➔ documentation de conception et de développement ;
  - ➔ documentation utilisateur ;
  - ➔ documentation technique de référence (→ Doxygen).
- Faire de l'intégration continue.
- Prévilégier des solutions multi-plateforme facilement déployables.
- Toujours penser à l'utilisateur bêta (d'ailleurs c'est toujours mieux de s'en trouver un très vite !).

3. DÉMARCHE POUR LA CONCEPTION

La différence qu'on pour voir entre un script et un logiciel c'est que le premier ne nécessite pas d'avoir une démarche de conception préalable (à part un minimum dans sa tête) alors que le second oui.

C'est quoi et comment ça marche la conception ?

1ère étape : la conception générale et les interfaces

C'est la définition des différents modules et/ou fonctionnalités du logiciel : qu'est-ce qu'ils font ? Quelles sont les principales variables ou fonctions de chaque module ? Comment est-ce qu'ils communiquent entre eux ? (interfaces internes) Et comment ils communiquent avec nous et les autres logiciels (interfaces externes). NB : les interfaces sont toujours difficiles à aborder dans tous les détails au début, elles nécessitent d'y revenir et de les affiner tout au long de la conception du logiciel (voir Illustration 1).

Pour consolider ces définitions on peut rajouter à cette étape un peu de ciment logiciel : comment est-ce qu'on gère les erreurs ? les logs ? les valeurs absentes ou invalides ? quel langage utilise t-on ? Etc.

Ensuite, on prend un peu de recul et on se pose la dernière question de la phase de conception générale : est-ce que toutes ces définitions semblent performantes et répondent au besoin ? Si oui, on continue. Si non : on recommence !

2ème étape : la conception détaillée

Evidemment on peut aller plus loin dans la conception, et si le logiciel est complexe ça sera nécessaire. Par contre, quand on vient de faire la conception générale précédente : on en a marre d'écrire et on voudrait que ça avance ! Et quand on a en marre, on ne fait pas du bon boulot, donc mieux vaut s'arrêter là pour le moment et commencer à programmer. On y reviendra plus tard au fil du développement pour écrire petit bout par petit bout la conception détaillée, qui probablement nous poussera de temps en temps à modifier légérement la conception générale.

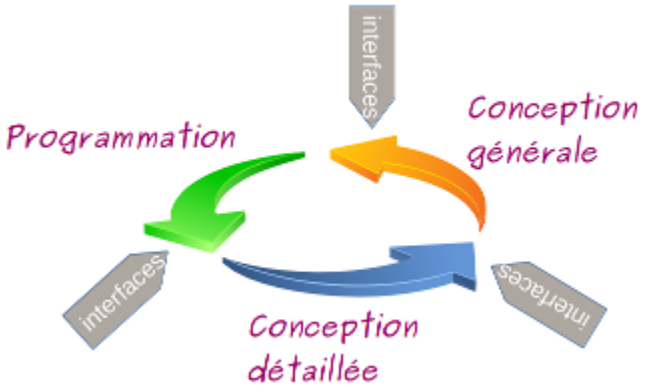

Illustration 1: Schéma de déroulement de la conception d'un logiciel.

Be Agile

Cette démarche de conception est inspirée des méthodes Agile.

Il y a une dizaine d'années, on ne faisait pas un logiciel avant d'avoir fait un document sur les

spécifications techniques, un autre sur les spécifications de réalisation, un autre sur les spécifications d'interfaces, un autre sur les spécifications techniques détaillées, etc. Le mot d'ordre était : tout est sous contrôle dès le départ ! Mais quand tout est sous contrôle… il n'y a plus de création. Et sans création, l'imagination se dégrade et les performances des logiciels aussi.

En 2001 est écrit le manifeste pour le développement Agile de logiciels : http://www.agilemanifesto.org/iso/fr/. Les méthodes Agile repensent le cycle de vie du logiciel, le rapport à utilisateur et le travail collaboratif. Un extrait du livre de Véronique Messager Rota (que je n'ai pas encore lu ; « Gestion de projet : vers les méthodes Agile », Broché, 2009) résume bien l'idée :

« Une méthode agile est une approche itérative et incrémentale, qui est menée dans un esprit collaboratif avec juste ce qu'il faut de formalisme. Elle génère un produit de haute qualité tout en prenant en compte l'évolution des besoins des clients ».

4. DOCUMENTATION NÉCESSAIRE

On favorise dans un développement les échanges directs avec les utilisateurs du logiciel et avec les autres développeurs. Néanmoins un certain niveau de documentation est nécessaire afin de synthétiser les réflexions, de s'en souvenir et de pouvoir fournir un support de formation pour les autres (et pour sois quand on aura tout oublié). Définir les documents nécessaires et leurs plans types est donc nécessaire à chaque début de projet.

De manière générale pour tout logiciel, on a définit les documents utiles suivants :

- **Le cahier des charges** : c'est le compte rendu des discussions avec l'utilisateur. Ca doit être synthétique et clair (sinon ça ne sera pas relu par l'utilisateur), reprendre le besoin (pour mémoire et être sûr qu'on s'est bien compris), et donner les grandes lignes de la solution proposée. On peut lui associer des spécifications algorithmiques si elles existent (faute de spécification algorithmique on s'appuiera souvent plutôt sur les protoypes existants et les articles scientifiques ou techniques associés).

  Plan type : 1/Contexte | 2/Expression du besoin | 3/Solution proposée.

- **Le document de développement** : il contient la description de la conception générale, des interfaces et de la conception détaillée. C'est le document central qui est écrit au fur et à mesure du développement du logiciel. Il est surtout orienté pour l'équipe de développement donc inutile d'en faire des tonnes, il faut que ça soit un document efficace, à chacun de juger le niveau de description qu'il doit y mettre.

  Plan type : 1/ Conception générale | 2/ Interfaces | 3/ Conception détaillée | 4/ Intégration des évolutions.

- **Le manuel de référence** : il référence tous les modules, classes et fonctions avec le descriptif de l'utilité et de l'utilisation de l'entité, des entrées/sorties, des limites possibles, etc. Un document très pratique mais aborder son écriture « à la main » est dangereux pour sa santé mentale. Heureusement Doxygen le fait pour nous de façon automatique et génére une documentation technique en HTML bien pratique. La seule chose à faire : bien structurer les commentaires du code en utilisant les balises Doxygen.

  Voir http://www.stack.nl/~dimitri/doxygen/.

- **Le document de validation** : on trace dans ce document la démarche de validation utilisée et l'évolution du logiciel. C'est à lui qu'on se réfère quand on cherche à savoir qu'est-ce qui a été testé et comment. Ce n'est pas non plus la peine d'en faire des tonnes (bien souvent mettre à disposition une fonction permettant de tester la valider d'une fonctionnalité est plus pertinente qu'un long discours). N'hésitez pas à préciser ce qui n'a pas été testé, les limites de la fonction qu'on a identifiées et celles dont on se doute. Le plus pratique est bien sûr d'écrire ce document au fur et à mesure du développement. C'est aussi un document qui va nous permettre d'assurer le suivi du logiciel : référencement des études de performances du logiciels qui seront réalisées dans le temps, des anomalies ou faiblesses identifiées.

  Plan type : 1/ Méthode de validation | 2/ Validation détaillée (un sous-chapitre pour chaque version-revision) | 3/ Suivi (un sous-chapitre pour chaque version-revision).

- **Le manuel d'utilisation** : c'est le seul document que l'utilisateur lira peut-être. Il a besoin d'un document court qui lui indique à quoi sert le logiciel, comment on l'installe, comment il fonctionne, quelles sont les limites, comment on l'utilise et qu'est-ce qu'il faut faire/prévoir pour le maintenir.

  Plan type : 1/ Introduction (description, licence et contact) | 2/ Installation | 3/ Fonctionnement et utilisation | 4/ Maintenance | ANNEXE – Exemples d'utilisation.

5. ORGANISATION DES FICHIERS SOURCES ET COMMENTAIRES

Organisation de fichiers sources :

- Un entête dans chaque fichier qui répond aux questions : qu'est-ce que c'est et à quoi ça sert ? qui a codé et quand? historique des évolutions ?
- Penser à préciser qu'il s'agit d'un encodage en utf8, c'est le langage universel il faut l'utiliser (surtout Python – en standard dans Python3).

Commentaires dans les codes et documentation technique de référence :

- Pour être tranquile avec les accents ne pas en mettre dans les commentaires. Et encore mieux : essayer de les écrire en anglais ça fait de l'exercice.
- Ne pas être radin avec les commentaires.
- Quoi de plus top quand on reprend un code que d'avoir une documentation technique de référence en ligne disponible (avec son équivalent pdf) et qu'il n'y ait plus qu'à cliquer sur un lien pour avoir le descriptif de la fonction, ses entrées/sorties et ses dépendances : **Doxygen** permet de faire ça rien qu'en structurant correctement ses commentaires.

  exemple : http://www.vtk.org/doc/nightly/html/index.html graphviz

  NB :

  1/ installer graphviz pour faire des jolies diagrammes (sudo apt-get install graphviz)

  2/ en python utiliser le filtre doxypy (sudo apt-get install doxypy), dans le fichier de configuration doxygen du projet :

  FILTER_SOURCE_FILES = YES

  INPUT_FILTER = "python /usr/bin/doxypy.py"

  OPTIMIZE_OUTPUT_JAVA = YES

  EXTRACT_ALL = YES

Un exemple d'utilisation de DOxygen dans du code python :

```
# -*- coding: utf-8 -*-


#- ----------------------------------------------------------------------
## @file logerr.py
## @defgroup logerr logerr                                    ←- pratique pour définir un module
## @brief Manage log, log file, traceback and exception
## @author Franck GABARROT,
## @date 2014
## @copyright CECILL-B
## @note
##- 2014/04/04: creation.
#- ----------------------------------------------------------------------


## @brief Available functions list to import
```

```
## @ingroup logerr                                    ←- et pour inclure dans le module
__all__=["genexcep","format_trace_from_sys","format_trace_from_genexcep",\
        "createLogger","debugmodeLogger","defaultmodeLogger",\
        "closeLogger","closeallLogger","changefileLogger",\
        "print2Logger"]

import os, sys, logging, logging.handlers, traceback

## @brief Module version (it is not the toolbox version).
## Integer value. Increment when you add an evolution, usefull to check compatibility.
## @ingroup logerr
__version__=1

#- -------------------------------------------------------------------
## @brief Create logger and create or open (if already exists) log file.
## @param[in] directory Log directory.
## @param[in] modulename Module name.
## @param[in] datetime String date and time information to add to the
##     filename : modulename_datetime.log. Default: no datetime string.
## @param[in] rotate Log file automatic rotating file greating than maxsize,
##     default: no rotating mode.
## @param[in] maxbytes Max file size for automatic rotating file (byte),
##     default: 1 Gbyte.
## @param[in] backupcount Max number of log files archived.
## @return Logger object from python-logging module. Return None if an
##     error occurs.
## @ingroup logerr
#- -------------------------------------------------------------------
def createLogger(directory, modulename, datetime=None, rotate=False, maxbytes=1000000, backupcount=10):
        """Create logger and create or open (if already exists) log file.
        """
        logger=None
```

… évidemment ce n'est pas très pythoniste de ne pas utiliser les docstring mais bon ...

En C++ on utilisera plutôt un formatage du style :

```
/**
 * @brief blablabla.
 * etc
 */
```

6. VERSIONNING ET ORGANISATION DES PAQUETS LOGICIELS

6.1 Le versionning

Format X.Y.Z :

- X : nombre entier correspondant à l'édition principale du logiciel. On l'incrémente lorsqu'on réalise une évolution majeure du logiciel (révolution!). X=0 est réservé à des versions prototypes ou encore instables et/ou non finalisées. X=1 est la première version stable.
- Y : nombre entier correspondant au numéro de révision. On l'incrémente quand on implémente des nouvelles fonctionnalités ou bien quand on les améliore, ou encore quand on corrige des grosses boulettes.
- Z : nombre entier correspondant à un numéro de correctif. C'est pour de la petite boulette.

On pourra rajouter à la version

- -alpha (1.0.0-alpha : on est sur la bonne voix de sortir la v1, il faut juste corriger les bugs et ajouter la carrosserie) ;
- -beta (1.0.0-beta : il faut tester à fond mais on est tout proche de la v1) ;
- -rc (1.0.0-rc, c'est la realease candidate, on est en v1 mais ça peut montrer qu'elle est toute jeune quand même). *Perso je ne suis pas fan de la nommenclature -rc, je la zappe.*

Autre type de versionning :

- Pour les **modules** : ça peut être utile d'avoir un versionning particulier pour les modules (test si cassure de rétro-compatibilité, etc). Par contre la nommenclature x.y.z ça embrouille avec le version du logiciel et c'est pas pratique à tester, donc pour les modules j'utilise plutôt W avec W un nombre entier de 1 à 54646465645644445.
- Pour les **documents** : c'est mieux d'avoir une nomenclature du document en plus de celle du logiciel : on associe la doc à une version X.Y du soft ou X.* mais entre temps elle peut aussi changer, donc elle va avoir une version L.Y avec L une lettre de a à z correspondant à l'édition de la doc et Y un numéro de révision classique.

6.2 Organisation standard

```
PaquetLogiciel/          ←- nom du logiciel
        bin/             ←- fichiers exécutables générés
        lib/             ←- librairies générées
        src/             ←- fichiers sources
        config/          ←- fichiers de configuration et/ou de paramétrage
        utils/           ←- utilitaires générés ou associés
        test/            ←- sources et executables de test
        doc/             ←- documentation
                algorithms/    ←- doc algo de référence
```

```
        development_manual/     ←- doc de conception, de développement, schémas, etc
            reference_guide/    ←- doc technique de référence (Doxygen)
                        html/       ←- dans sa version html
                user_guide/     ←- doc utilisateur
                validation/     ←- doc de validation et de suivi des anomalies
    AUTHOR      ←- fichier contenant le ou les noms des auteurs
    CHANGELOG   ←- fichier traçant les modifications
    LICENSE     ←- fichier contenant la licence (+ LICENSE-fr)
    README      ←- description du paquet logiciel (référence à la documentation  dans
                   doc s'il y en a)
    INSTALL     ←- notes pour l'installation (référence à la documentation utilisateur
                   s'il y en a une)
    TODO        ←- qu'est-ce qu'on doit encore faire et/ou prévu de faire
    VERSION     ←- version du paquet x.x.x
    install.sh, etc ←- fichiers d'installation si on est chaud
```

6.3 Adaptation pour Python

La particularité de python est qu'en plaçant un fichier __init__.py dans un répertoire on crée un espace de nom avec le nom du répertoire, donc on va éviter 'src'.

```
PaquetLogiciel/         ←- nom du logiciel
    bin/        ←- fichiers exécutables générés
    lib/        ←- librairies générées
    paquetlogiciel/         ←- fichiers sources (en miniscule) dont __init__.py
    config/     ←- fichiers de configuration et/ou de paramétrage, on peut aussi
                   mettre se répertoire dans le répertoire paquetlogiciel, en python
                   c'est plus logique.
    utils/      ←- utilitaires générés ou associés
    test/       ←- sources et executables de test
    doc/        ←- documentation
      algorithms/   ←- doc algo de référence
      development_manual/   ←- doc de conception, de développement, schémas, etc
      reference_guide/  ←- doc technique de référence (Doxygen)
                html/       ←- dans sa version html
      user_guide/   ←- doc utilisateur
      validation/   ←- doc de validation et de suivi des anomalies
    AUTHOR      ←- fichier contenant le ou les noms des auteurs
```

CHANGELOG ←- *fichier traçant les modifications*

LICENSE ←- *fichier contenant la licence (+ LICENSE-fr)*

README ←- *description du paquet logiciel (référence à la documentation dans doc s'il y en a). On peut faire un README.md (format ReST) qui est plus pythoniste si on est chaud.*

INSTALL ←- *notes pour l'installation (référence à la documentation utilisateur s'il y en a une)*

TODO ←- *qu'est-ce qu'on doit encore faire et/ou prévu de faire*

VERSION ←- *version du paquet x.x.x*

setup.py ←- *fichier d'installation Python*

MANIFEST.in ←- *fichier texte qui liste les fichiers non python à inclure dans l'installation*

7. UTILISATION D'UNE LICENCE

Il ne faut pas diffuser un code qui n'a pas de nom d'auteur, de date et de licence associée… on ne sait jamais ce dont demain est fait et après tout c'est notre travail alors autant le valoriser.

Le plus simple c'est l'ensemble des licences CECILL élaborées par le CNRS, l'INRIA et le CEA afin de développer des logiciels libres de droit français dans l'esprit GNU GPL.

Voir http://www.cecill.info/.

8. VALIDATION DES CODES ET INTÉGRATION CONTINUE

- Faire un document de validation pour assurer le suivi et se rappeler des points qu'on a testés et ceux qu'on n'a pas testés. Ce document permet de garder une trace de l'évolution des logiciels : il est important.
- Faire des codes de test pour chaque fonction développée et surtout dans les conditions limites (à placer dans le répertoire test). Au minimum pour les fonctions complexes.
- Faire de l'intégration continue. Sans déballer la grosse artillerie mais juste s'assurer que la fonction développée peut partir dans la nature et être utilisée par une personne tierce. S'il s'agit de la modification d'une fonction existante, s'assurer que les fonctions qui l'utilisent ont toujours le même comportement (rejouer les tests de ces fonctions et contrôler leur validité).

  Un truc pratique pour ça utilisant Doxygen : pour chaque fonction qui utilise une autre fonction ajouter la balise @sa (après la balise @brief c'est plus jolie) suivi du nom complet de la fonction. Ca lie les fonctions dans le document de référence et on peut faire une recherche dans les fichiers avec '@sa *nom-fonction*' afin de s'assurer qu'on a vérifié toutes les autres fonctions qui l'utilise.

  Exemple :

```
## ------------------------------------------------------------------------
## @brief Build a XML tree from a XML-like dictionnary. Recursive function.
## @sa miscbox.mstring.compact_values_to_str()
## @param[in] dic2write XML-like input dictionnary or list.
## @param[in] roottagname Main XML tag name.
## @param[in] rootelement Main XML element to complete with tree.
## @param[in] rootwrite Build XML tree from main tag (default=True).
## @return 1 if an error occurs, 0 elsewhere.
## @ingroup mxml
## ------------------------------------------------------------------------
def dic2Xml(dic2write, roottagname, rootelement, rootwrite=True):
        [...]
```

9. FORMALISME PRÉVILÉGIÉ POUR LES APPELS ET LES RETOURS DES FONCTIONS [PYTHON]

9.1 Structure d'appel d'une fonction

On identifie en fonction du nombre d'arguments 3 catégories de fonctions :

- **L** (low) : en dessous de 6 arguments.

  function(arg1, arg2, …, argm=argm, argn=argn)

  RAS c'est du classique

- **M** (medium) : au dessus de 6 arguments.

  function(arg1, arg2, …, **kwargs)

  Le mot clé « args » permet un nombre d'arguments positionnels indéfini. Le mot clé **kwargs permet un nombre d'arguments nommés indéfini. Ils sont utiles si on veut faire évoluer la structure d'appel de la fonction tout en assurant la rétro-compatibilité. En pratique on va éviter les « args » et on va surtout surtout utiliser les « **kwargs » pour éviter une structure d'appel à rallonge et se laisser la possibilité de rajouter des options. Attention tout de même à ne pas en abuser, ce n'est pas des plus lisibles pour l'utilisateur.

- **H** (high) : un grand nombre (fonction fortement configurable) et adapté pour les fonctions d'interface.

  function(configfile=configfile, **kwargs)

  | **|**

  function_v1(params, **kwargs) **function_v2(params, **kwargs)**

  La fonction d'interface utilise un fichier de configuration contenant la liste des paramètres et la version de la fonction à appeler, les **kwargs contiennent la liste des objets et autres variables hors paramètres de configuration à manipuler. La fonction d'interface exploite un parser du fichier de configuration qui fournit un dictionnaire « params » contenant l'ensemble des paramètres attendus et la version de la fonction.

9.2 Retour d'une fonction

Si la fonction est simple (pas beaucoup de cas d'erreur à gérer), classiquement on renvoit une valeur ou un dictionnaire de valeurs en retour et None en cas d'erreur si on attend un retour sinon un code de retour.

valeur ou dictionnaire de valeurs (None sur erreur) ou **code retour**

Si la fonction est plus complexe, on voudra savoir à quel endroit l'erreur s'est produite grace à un message d'erreur précis, on préviligiera le tuple suivant en retour :

(valeur ou dictionnaire de valeurs (None sur erreur), code retour, message d'erreur)

ou (**code retour, message d'erreur)**

Code retour :

- 0 : tout s'est bien déroulé.
- 1 : erreur fatale.
- 2 : alerte.
- 1xx : erreur fatale – code spécialisé.
- 2xx : alerte – code spécialisé.

  NB : en pratique on évite l'utilisation d'un code spécialisé sauf nécessité absolue.

10. BRIC À BRAC [C++]

10.1 Cmake

TODO

10.2 C++11

TODO

10.3 Boost.Python

TODO

11. BRIC À BRAC [PYTHON]

11.1 Flottants : précision et affichage

Précision :

Les flottants (en base 10) sont codés en base 2. Conséquence : le codage en base 2 stocké dans l'ordinateur est une approximation de son homologue en base 10.

Voir http://www.afpy.org/doc/python/2.7/tutorial/floatingpoint.html ou encore http://en.wikipedia.org/wiki/Machine_epsilon.

Mais pas de panique, maintenant on n'est que sur des systèmes 64bits, le type float python est un float 64, pour vérifier :

```
import sys
sys.float_info
sys.float_info(max=1.7976931348623157e+308,max_exp=1024,max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021,min_10_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

Si on utilise numpy :

```
>> import numpy as np
Precision :
>>finfo32 =np.finfo(np.float32)  >>print(finfo32)  →1e-6  (environ 7 chiffres significatifs)
>>finfo64 =np.finfo(np.float64)  >>print(finfo64)  →1e-15 (environ 16 chiffres significatifs)
Tableaux :
>>a=np.array([[1,2,3], [4,5,6]], dtype('float64'))
>>a.dtype()
dtype('float64')
Scalaires :
>>x = np.float64(25.2525)
>>x.dtype   (ou type(x))
<type 'numpy.float64'>
NB : le type par défaut de Numpy sur archi 64 bits c'est du float64.
```

Affichage et précision :

```
print("{0:.2f}".format(your_number))
```

11.2 L'accès aux librairies de la communauté python: http://pypi.python.org/pypi

sudo apt-get install python-setuptools

easy_install --user pip

puis il n'y a plus qu'à faire pip install --user librairie et c'est gagné.

Pip fait aussi les désinstallation, les mises à jour, etc.

Liste de toutes les libs installées : pip freeze.

Donc pour recréer son environnement : pip freeze > requirements.txt puis pip install -r requirements.txt

11.3 Packaging

11.3.1 __all__

Si on a un module dans lequel on veut que certaines fonctions ne soient pas accessibles (internes), on utilise __all__=['fonction1','fonction2','parametre1'] qui va limiter l'importation à ces fonctions.

11.3.2 __init__.py

Dans ce fichier on peut mettre une docstring avec le descriptif du paquet et surtout sa version :

__version__="0.0.1"

11.3.3 Gérer l'environnement

Surtout quand on développe, on n'a pas envie de mixer ses librairies tout de suite avec celles de python qui marchent bien dans /usr/lib/python2.7/dist-packages :

- soit on fait un script Bash qui fait adapte la variable d'environnement PYTHONPATH qu'on va exécuter à chaque fois (ou qu'on place dans le .bashrc) :

  export PYTHONPATH=$PYTHONPATH:/home/moi/meslib2 :home/moi/meslib2
- soit on utilise virtualenv : pip install --user virtualenv.

11.3.4 setup.py

C'est chiant mais c'est magique en utilisant setuptools et un fichier setup.py. Voilà un bon exemple pour une lib ayant pour nom sm_lib (source ST02) :

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from setuptools import setup, find_packages

# notez qu'on import la lib
# donc assurez-vous que l'importe n'a pas d'effet de bord
import sm_lib

# Ceci n'est qu'un appel de fonction. Mais il est trèèèèèèèèèès long
# et il comporte beaucoup de paramètres
setup(

    # le nom de votre bibliothèque, tel qu'il apparaitre sur pypi
    name='sm_lib',

    # la version du code
    version=sm_lib.__version__,
```

```
# Liste les packages à insérer dans la distribution
# plutôt que de le faire à la main, on utilise la foncton
# find_packages() de setuptools qui va cherche tous les packages
# python recursivement dans le dossier courant.
# C'est pour cette raison que l'on a tout mis dans un seul dossier:
# on peut ainsi utiliser cette fonction facilement
packages=find_packages(),

# votre pti nom
author="Sam et Max",

# Votre email, sachant qu'il sera publique visible, avec tous les risques
# que ça implique.
author_email="lesametlemax@gmail.com",

# Une description courte
description="Proclame la bonne parole de sieurs Sam et Max",

# Une description longue, sera affichée pour présenter la lib
# Généralement on dump le README ici
long_description=open('README.md').read(),

# Vous pouvez rajouter une liste de dépendances pour votre lib
# et même préciser une version. A l'installation, Python essayera de
# les télécharger et les installer.
#
# Ex: ["gunicorn", "docutils >= 0.3", "lxml==0.5a7"]
#
# Dans notre cas on en a pas besoin, donc je le commente, mais je le
# laisse pour que vous sachiez que ça existe car c'est très utile.
# install_requires= ,

# Active la prise en compte du fichier MANIFEST.in
include_package_data=True,

# Une url qui pointe vers la page officielle de votre lib
url='http://github.com/sametmax/sm_lib',

# Il est d'usage de mettre quelques metadata à propos de sa lib
```

```
    # Pour que les robots puissent facilement la classer.
    # La liste des marqueurs autorisées est longue, alors je vous
    # l'ai mise sur 0bin: http://is.gd/AajTjj
    #
    # Il n'y a pas vraiment de règle pour le contenu. Chacun fait un peu
    # comme il le sent. Il y en a qui ne mettent rien.
    classifiers=[
        "Programming Language :: Python",
        "Development Status :: 1 - Planning",
        "License :: OSI Approved",
        "Natural Language :: French",
        "Operating System :: OS Independent",
        "Programming Language :: Python :: 2.7",
        "Topic :: Communications",
    ],

    # C'est un système de plugin, mais on s'en sert presque exclusivement
    # Pour créer des commandes, comme "django-admin".
    # Par exemple, si on veut créer la fabuleuse commande "proclame-sm", on
    # va faire pointer ce nom vers la fonction proclamer(). La commande sera
    # créé automatiquement.
    # La syntaxe est "nom-de-commande-a-creer = package.module:fonction".
    entry_points = {
        'console_scripts': [
            'proclame-sm = sm_lib.core:proclamer',
        ],
    },

    # A fournir uniquement si votre licence n'est pas listée dans "classifiers"
    # ce qui est notre cas
    license="WTFPL",

    # Il y a encore une chiée de paramètres possibles, mais avec ça vous
    # couvrez 90% des besoins
)
```

Ya plus qu'à : python setup.py install